SPARTACAS: Automating Component Reuse and Adaptation

Brandon Morel and Perry Alexander, *Senior Member*, *IEEE*

Abstract—A continuing challenge for software designers is to develop efficient and cost-effective software implementations. Many see software reuse as a potential solution; however, the cost of reuse tends to outweigh the potential benefits. The costs of software reuse include establishing and maintaining a library of reusable components, searching for applicable components to be reused in a design, as well as adapting components toward a proper implementation. In this paper, we introduce SPARTACAS, a framework for automating specification-based component retrieval and adaptation that has been successfully applied to synthesis of software for embedded and digital signal processing systems. Using specifications to abstractly represent implementations allows automated theorem-provers to formally verify logical reusability relationships between specifications. These logical relationships are used to evaluate the feasibility of reusing the implementations of components to implement a problem. Retrieving a component that is a complete match to a problem is rare. It is more common to retrieve a component that partially satisfies the requirements of a problem. Such components have to be adapted. Rather than adapting components at the code level, SPARTACAS adapts the behavior of partial matches by imposing interactions with other components at the architecture level. A subproblem is synthesized that specifies the missing functionality required to complete the problem; the subproblem is used to query the library for components to adapt the partial match. The framework was implemented and evaluated empirically, the results suggest that automated adaptation using architectures successfully promotes software reuse, and hierarchically organizes a solution to a design problem.

Index Terms—Reuse models, formal methods, programmer workbench, reuse library.

1 Introduction

As engineers continue to struggle with cost and time associated with software development, reuse has emerged as a sound engineering principle and practice in many design fields. Some engineering disciplines, such as hardware engineering, have seen the proliferation of commercial off-the-shelf components that can be used to successfully construct systems. The introduction of repositories and libraries, either within organizations or globally via the World Wide Web, of reusable software components has helped forward the application of reuse into the software engineering realm.

Software reuse [7], [17], [18] potentially has many alluring benefits, including the ability to increase the productivity of engineers, reduce errors early in system design, and increase the quality and reliability of software produced. However, for software reuse to become widespread, its benefits must outweigh its costs. These costs include the effort to create and maintain a library of reusable components, and the costs associated with retrieving, adapting, and integrating reusable components into an implementation to a design problem.

While successful experiments are becoming more common, the practice of software reuse has been slow to realize the potentials claimed by its advocates. Several works [8], [15] attribute the unfulfilled promise to various technical and nontechnical reasons. While nontechnical issues are unavoidable, one technical obstacle still remains to be resolved, namely, automating the adaptation of components. Many works [12], [4], [10], [27] have successfully developed automated software component retrieval systems and frameworks at almost every level of software development. In these frameworks, the aim is to retrieve a component from a library that is a complete match to a particular problem. It is rare for a software library to always contain a perfect match to a problem. It is more feasible to find a similar component and adapt it to satisfy the problem. Although automating adaptation has long been an area of research in case-based reasoning and knowledge-based systems [29], [16], few experiments have attempted to address the issue of software adaptation [22], [11].

Safely and correctly modifying software at the code level is not always a trivial task [19]. Adapting complex software code may not outweigh the benefits of producing the software from scratch. Penix [22], [23] proposed adapting software components at the architecture level. By imposing interactions with other components in an architecture, the behavior of a partial match is adapted in such a way that, for all given inputs, the output of the architecture satisfies the requirements of the problem. In addition to increasing the potential of software reuse, adaptation architectures also provide an organizational design hierarchy.

In this paper, we show that adaptation can be automated at the specification level using adaptation architectures. We have developed a framework for reuse, known as SPecification-based Architecture and Retrieval Techniques for Automating Component Adaptation and Synthesis (SPARTACAS), and have successfully applied it to the synthesis of software for embedded and digital signal processing

• *The authors are with the Information and Telecommunication Technology Center, University of Kansas, 2335 Irving Hill Rd., Lawrence, Kansas 66045. E-mail: {morel, alex}@ittc.ku.edu.*

Manuscript received 21 Aug. 2003; revised 20 Jan. 2004; accepted 2 July 2004. Recommended for acceptance by B. Frakes.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0115-0803.

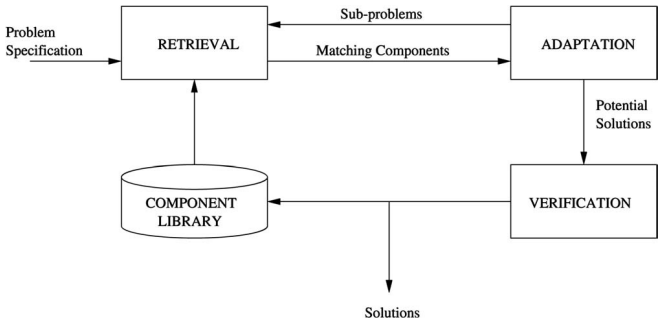


Fig. 1. General component reuse framework.

```
package componentName() :: domainName is
  export all;
begin

  facet componentName(parameterList) :: domainName is
    export all;
  begin
    termLabel: term;
    ...
  end facet componentName
end package componentName
```

Fig. 2. Specification structure in Rosetta.

systems. The framework uses a specification-based retrieval engine to retrieve software components that are either complete or partial matches to a problem. If a complete match cannot be found, a subproblem that specifies the missing functionality required to solve the problem is synthesized. The subproblem is used to query the library for components to adapt the partially matching component in an architecture. We will begin with an overview of the framework, then focus on the adaptation architectures and the tactics for synthesizing subproblems for adaptation. We will conclude by experimentally evaluating our framework and discuss future and related work.

2 Component Reuse Framework

The primary objective of SPARTACAS is to retrieve possible solutions to a problem from a library of components. In addition to retrieval, the SPARTACAS framework, shown in Fig. 1, includes an automated adaptation capability. Given a problem specification, a retrieval engine returns both total and partial matches. For partial matches, the behavior is adapted by imposing interactions with other components in an adaptation architecture. A subproblem that specifies the missing functionality required to solve the rest of the problem is synthesized and used to search for components to satisfy the architecture. The architectural solution is verified and added to the component library, further increasing the potential of reuse.

2.1 Component and Problem Specifications

Each component implementation includes a formal specification that states the behavior of a component without stating the implementation details. Using formal specifications over implementations allows automated theorem-provers to verify match conditions between two components. The formal component specifications use a simple axiomatic structure [13], [28]:

$$\forall d \in D, \exists r \in R | I(d) \Rightarrow O(d, r).$$

D and R are the domain (input types) and range, respectively. The domain represents the input values to the component and the range represents the output values of the component. I is a set of preconditions that define the *legal inputs* to the component. The preconditions constrain the domain to the values that have a defined output. O is a set of postconditions that define the *feasible outputs* for each legal input based on a D $\times$ R relation. If the preconditions hold then the component will end in a state such that the postconditions are true. If the preconditions do not hold, then there are no guarantees that the postconditions will hold, however, termination is assumed in all cases.

The component specifications are written in Rosetta [1]. Rosetta is a systems level design language for modeling heterogeneous systems. A Rosetta *facet* (not to be confused with a facet used in Prieto-Diaz and Freeman's work [25]) describes the requirements or behavior of a particular aspect of a system or component. Facet parameters declare the domain (*input* typed variable declarations) and range (*output* typed variable declarations) of components. A facet operates in a declared domain, which defines the vocabulary of semantics available to the facet. We use term labels starting with *pre* and *post* to define the pre and postconditions over the inputs and outputs, respectively. Term labels that start with *arch* will be used to define structural component specifications and structural solutions to problems. The terms that can be expressed in Rosetta are similar to those that can be expressed in most functional languages. The structure of the specifications in Rosetta is shown in Fig. 2.

A component library contains a collection of existing component specifications that have been created, tested, and shown to correctly specify the component's functionality. A problem specification is a representation of a design that does not yet have an implementation. A problem specification is written using the same model as a component. The goal will be to retrieve components that match a problem, implying that the implementation of the component can be reused to implement the problem.

2.2 Component Retrieval Framework

The retrieval framework, see Fig. 3, contains a collection of *features*, which are a set of theorems that capture some concept within a field of knowledge. The following is an example of a feature that filters elements from a list:

$$\begin{aligned} &FILTER(list, element) \\ &\equiv \exists x, y : list | \forall z : element | (z \in y \Rightarrow z \in x) \\ &\quad \wedge (x \in D) \wedge (y \in R). \end{aligned}$$

A specification is assigned a feature if it can be proven from the specification. The features assigned to the components in the library are stored in a database, called a *featurebase*. Feature classification of the components in the library is performed offline, only feature classification of a

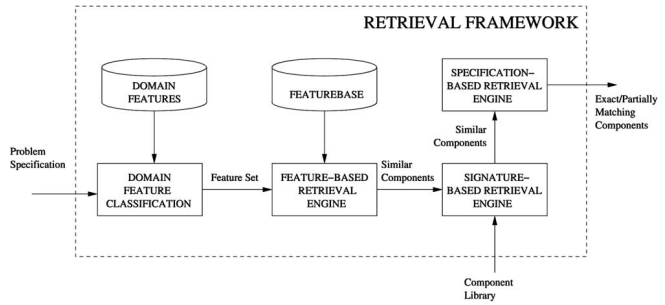


Fig. 3. Component retrieval framework.

problem specification is perform during retrieval. The *feature-based retrieval engine* filters out all of the components that do not have the same feature classification.

The component search space is reduced by the feature-based retrieval engine and given to the *signature-based retrieval engine* [31]. A signature represents the input and output types, it does not contain any semantic information about the component or problem. The signature-based retrieval engine filters out components that do not have compatible signatures. Components that do have compatible (exact or relaxed) signatures can be instantiated to possibly solve the problem. The information used by the signature-based retrieval engine is also used in the instantiation of components in an architecture.

The last layer is the *specification-based retrieval engine*, which performs specification matching [14]. Several specification-based retrieval engines [6], [24] have been developed using automated theorem-provers to logically verify that a component specification matches a problem specification. Zaremski and Wing [32] established a number of match conditions (sometimes referred to as the *degree of a match* or *degree of satisfaction*) for assessing reuse. Fig. 4 shows a portion of these match conditions.

If a component C formally *Satisfies* a problem P, then the implementation of C can be reused to implement P. C satisfies P if C accepts all legal inputs to P, and the valid outputs of C are valid outputs of P when given legal inputs. *Weak Plug-in* and *Plug-in* are stronger match conditions of Satisfies. The *Plug-in Pre* match condition implies that a

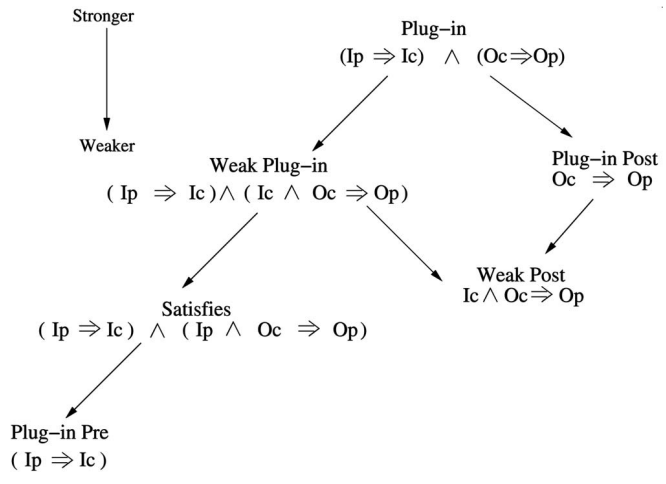


Fig. 4. Specification match lattice.

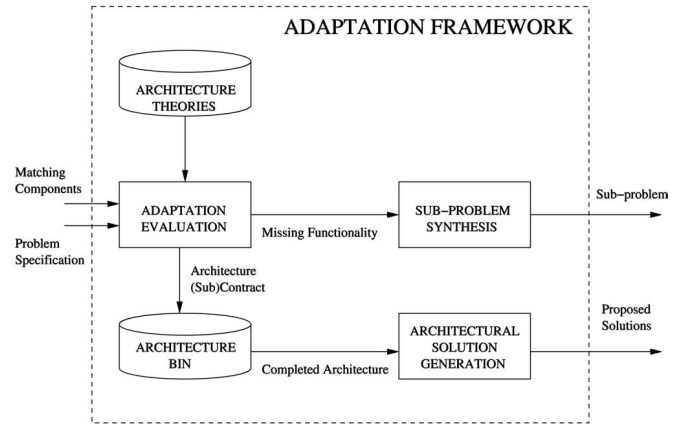


Fig. 5. Component adaptation framework.

component meets only the precondition requirements. The *Plug-in Post* and *Weak Post* match conditions imply that a component meets the postcondition requirements, but do not have the proper preconditions. Plug-in Pre, Plug-in Post, and Weak Post are referred to as *partial match conditions*.

2.3 Component Adaptation Framework

The adaptation framework, see Fig. 5, contains a collection of adaptation architecture theories. They specify the constraints on the interconnection of components, the effects of adaptation on the behavior of a component, and the functionality of the overall system using instantiated components. Given a partially matching component to a problem, the *adaptation evaluation* module determines which adaptation architecture can be applied. It is possible that several adaptation architectures are applicable. The architecture construction tactic and the component is added to the *architecture bin* as a "contract." The architecture bin acts as a tree of architectural blueprints, which is used to plan the execution of contracts toward a solution.

Based on the information given in the component, problem, and architecture specifications, the missing functionality required to solve the rest of the problem is synthesized. The *synthesizer* generates a subproblem specification that is resubmitted to the retrieval engine. The subproblem may also generate architectural-subcontracts. A component that completely satisfies a problem (or subproblem) signifies that an architectural contract (or subcontract) has been completed. Adaptation continues until 1) an architecture of interconnected components satisfy the problem constraints, 2) a solution to the design problem is known not to exist given the current reuse library, or 3) a solution to the design problem can not be realized with the current retrieval and adaptation process, e.g., search depth limitations.

The components used in this work are derived from the DSP/control and mathematical knowledge fields. The reuse methodology can be applied to various domains, given that knowledge-field specific features are defined for the feature-based retrieval engine, the adaptation architectures are valid in the domain, and components in the domain can be represented using the DRIO model.

TABLE 1
Available Adaptation Architecture Tactics

Architecture	*Match Condition*	*Instantiation*
Sequential	Plug-in Post Weak Post	Plug component into tail position, derive sub-problem to satisfy head position
	Plug-in Pre	Plug into head position, derive sub-problem to satisfy tail position
Alternative	Weak Post	Plug into either position, derive sub-problem to satisfy missing functionality
Parallel	N/A	Problem decomposition, independent instantiation

Sequential Architecture Theory
BEGIN

// Problem and component definitions
Problem(D, R, I, O)
Component$_A$(D$_A$, R$_A$, I$_A$, O$_A$)
Component$_B$(D$_B$, R$_B$, I$_B$, O$_B$)

// Domain and range constraints
drConstraint1: D $\subseteq$ D$_A$
drConstraint2: R$_A$ $\subseteq$ D$_B$
drConstraint3: R$_B$ $\subseteq$ R

// Pre and postcondition constraints
behConstraint1: $\forall$ d : D | I(d) $\Rightarrow$ I$_A$(d)
behConstraint2: $\forall$ d : D, x : D$_B$ | I(d) $\wedge$ O$_A$(d, x) $\Rightarrow$ I$_B$(x)
behConstraint3: $\forall$ d : D, y : R$_A$, r : R | I(d) $\wedge$ O$_A$(d, y)
$\wedge$ O$_B$(y, r) $\Rightarrow$ O(d, r)

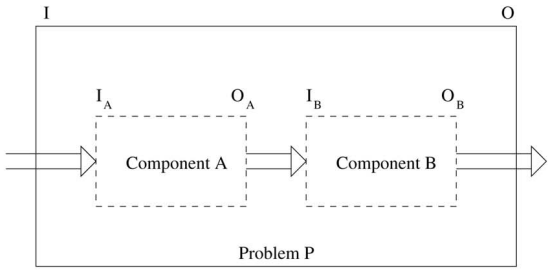


Fig. 6. Sequential architecture theory.

3 Adaptation Architecture Theories

Behavioral adaptation is the process of adding or limiting the functionality of a component by imposing interactions with other components in an architecture, such that the architecture provides the desired functionality. An *adaptation architecture* is a formal specification that specifies the interaction and configuration of subcomponents in the composition of a system and also specifies the relationship between the functionality of the subcomponents and the functionality of the system. Penix [23] proposed three adaptation architecture theories: sequential, alternative, and parallel. One or more of these adaptation architectures can be applied to adapt the behavior of a partially matched component. The degree to which a component satisfies a problem determines which adaptation architecture tactic is applied. Table 1 shows the adaptation tactics associated with each match condition.

The sequential architecture, Fig. 6, is the interconnection of two components where the output of one component is the input to another component through some type of communication medium. The head component adapts the input values to the tail component such that the tail component generates feasible outputs. Conversely, the tail component adapts results generated by the head component for all legal inputs. Double arrows represent the collection of interconnected ports. A *port* is an input or output typed variable. During retrieval, the signature-matching engine attempts to find components with matching signatures. Fig. 6 also specifies constraints over the types of interconnected ports. The left of Fig. 6 lists several constraints that specify the relationships between the functionality of the problem and the functionalities of the subcomponents.

The alternative architecture is the interconnection of two independent components that work simultaneously and their outputs are combined to satisfy the problem requirements (shown in Fig. 7). In this architecture, one component satisfies the problem for some subset of legal input values to the problem. This component is adapted with another component which correctly covers the rest of the input values. This architecture requires a control structure to achieve correct cooperation. Without a control structure, the two components could possibly diverge on a given input and drive contradictory results on the output.

The parallel architecture, Fig. 8, is similar to the alternative architecture. Independent components work simultaneously and their outputs collectively satisfy the problem requirements. The difference is that the output values generated by the components in the parallel architecture are not combined to affect a single output. The components in the parallel architecture compute on disjoint subranges, which collectively form the range of the problem. Each component computes results for some (not necessarily disjoint) subset of the inputs of the problem. Fig. 8 uses the $||$ notation to represent the decomposition of some set of the range into disjoint subranges. For instance, a problem may have the following output ports: $\{\mathrm{x}::\mathrm{integer}, \mathrm{y}::\mathrm{boolean}, \mathrm{z}::\mathrm{real}\}$. The range could possibly be represented as: $\{\mathrm{y}::\mathrm{boolean}\}||\{\mathrm{x}::\mathrm{integer}, \mathrm{z}::\mathrm{real}\}$. The parallel adaptation architecture decomposes a problem into independent subproblems. Components that satisfy these subproblems never interact with each other, they merely solve an isolated aspect of the problem.

Alternative Architecture Theory
BEGIN

// Problem and component definitions
Problem(D, R, I, O)
$\text{Component}_A(\text{D}_A, \text{R}_A, \text{I}_A, \text{O}_A)$
$\text{Component}_B(\text{D}_B, \text{R}_B, \text{I}_B, \text{O}_B)$

// Domain and range constraints
drConstraint1: $\text{D} \subseteq \text{D}_A$
drConstraint2: $\text{D} \subseteq \text{D}_B$
drConstraint3: $\text{R}_A \subseteq \text{R}$
drConstraint4: $\text{R}_B \subseteq \text{R}$

// Pre and postcondition constraints
behConstraint1: $\forall d : D | (I(d) \Rightarrow I_A(d)) \vee (I(d) \Rightarrow I_B(d))$
behConstraint2: $\forall d : D, r : R | (I_A(d) \wedge O_A(d, r) \Rightarrow O(d, r))$
$\vee (I_B(d) \wedge O_B(d, r) \Rightarrow O(d, r))$

END Alternative Architecture Theory

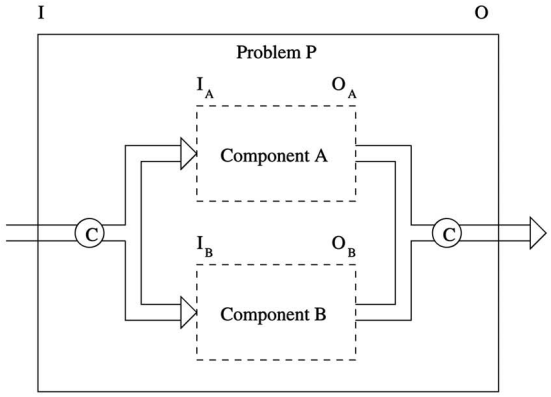


Fig. 7. Alternative architecture theory.

Parallel Architecture Theory
BEGIN

// Problem and component definitions
Problem(D, R, I, O)
$\text{Component}_A(\text{D}_A, \text{R}_A, \text{I}_A, \text{O}_A)$
$\text{Component}_B(\text{D}_B, \text{R}_B, \text{I}_B, \text{O}_B)$

// Domain and range constraints
drConstraint1: $\text{D} \subseteq \text{D}_A \cup \text{D}_B$
drConstraint2: $\text{R}_A \,||\, \text{R}_B \subseteq \text{R}$

// Pre and postcondition constraints
behConstraint1: $\forall d_1 \cup d_2 : D | (I(d_1 \cup d_2) \Rightarrow I_A(d_1) \wedge I_B(d_2)$
behConstraint2: $\forall d_1 \cup d_2 : D, r_1 || r_2 : R | I(d_1 \cup d_2) \wedge O_A(d_1, r_1)$
$\wedge O_B(d_2, r_2) \Rightarrow O(d_1 \cup d_2, r_1 || r_2)$

END Parallel Architecture Theory

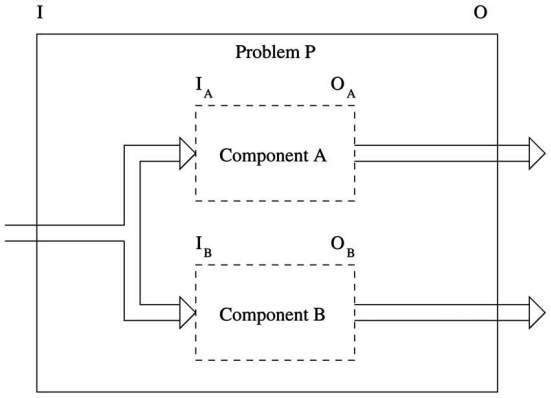


Fig. 8. Parallel architecture theory.

The three adaptation architectures can be used to compose increasingly complex architecture structures. A component declared in an adaptation architecture theory specification can abstractly represent other architectures. Moreover, components in the library can also represent user-defined or knowledge field-specific architectures which can be retrieved and instantiated in other architectures.

4 Port Connection Methods

Component and problem specifications define a input and output variables through port declarations. A port is an input/output typed variable that is referenced in the pre and postconditions that constrain the legal input and valid output values. Much of the early work in software retrieval dealt with signature matching [31], where a signature is comprised of input parameters and return types of functions, procedures, and other such software artifacts. In specification-based matching, the ports of a component are instantiated with the ports of a problem, which requires that the component and problem have compatible input and output ports. The instantiation of ports, or port connection, is given in Definition 1. Operations such as currying, generalization, and specialization [31] of types can be applied to find a proper mapping.

Definition 1. *A port connection ρ is a function mapping the ports of component $C(D_C, R_C, I_C, O_C)$ to the ports of problem $P(D, R, I, O)$ such that:*

- $\forall$ *input port* $c_i : T_c \in \mathbf{D_C} |$
  $\exists$ *input port* $p_i : T_p \in \mathbf{D} | (\rho(c_i) \rightarrow p_i) \wedge (T_p \subseteq T_c)$.
- $\forall$ *output port* $c_o : T_c \in \mathbf{R_C} |$
  $\exists$ *output port* $p_o : T_p \in \mathbf{R} | (\rho(c_o) \rightarrow p_o) \wedge (T_c \subseteq T_p)$

Bijective port connection retrieves components to problem for which there is a one-to-one and onto mapping from input ports of the component to the input ports of the problem and similarly a mapping of component output ports to problem output ports. In traditional signature matching, the component and problem must have an equal number of compatible input ports and output ports. This can hinder potential applications for adaptation. Fig. 9 specifies a problem to perform simple addition on two real numbers. Fig. 10 shows a successfully instantiated solution using the simple mathematical operations library in Fig. 13. Following retrieval, it is determined that the numerical subtraction (*sub*)component is a (bijective) Plug-in Prepartial match to the problem. The numerical negation (*negate*) component is not retrieved since it does not have the proper number of ports for instantiation. Clearly, a solution cannot

```
// Simple addition problem
package P1() :: null is
  export all;
begin
  facet P1(a :: input real; b :: output real;
           c :: output real) :: state_based_semantics is
    export all;
  begin
    pre:   true;
    post:  c' = (a + b);
  end facet P1;
end package P1;
```

Fig. 9. Problem specification of a simple addition problem.

be found using bijective signature matching for even the simplest problems, thus motivating the need for less restrictive port connection methods.

A less restrictive port connection method is the *one-to-one port connection*. The one-to-one port connection requires that all component input ports be driven by one and only one problem input port, and all component output ports drive one and only one problem output port. This allows potential components to have fewer number of input/output ports than the problem. However, since not all the ports of the problem can be instantiated, components that are retrieved during component retrieval will not completely satisfy the problem (assuming the uninstantiated ports have meaningful constraints on them). A subproblem is required to search for other components to instantiate and satisfy the functionalities of the unconnected ports.

The *onto port connection* method simply requires that, for all ports in the component, there is a connection to some problem port with respect to port direction. Potentially a single problem input port can drive multiple component input ports, and a single problem output port can be driven by multiple component output ports (in such a case the final output value needs to be resolved). This allows the component to have more ports than the problem.

Although this section motivated using less restrictive port connection methods to increase recall, doing so may result in expensive overhead. The number of instantiations per component may drown the specification-based retrieval engine from making timely progress.

5 Automated Component Adaptation

Given a partial match to a problem, the problem and partial match can be instantiated into an adaptation architecture. The adaptation theory used is contingent on the match condition between the problem and the partially matched component. Using the relationship between the functionalities of the subcomponents and the functionality of the system defined in the adaptation architecture, the missing functionality required to satisfy the rest of the problem can be synthesized into a subproblem. The subproblem represents the components needed to adapt the partial match.

5.1 Sequential Adaptation

Recall from Table 1 that Plug-in Post, Weak Post, and Plug-in Prematched components can be adapted using the sequential adaptation architecture. The missing functionality synthesized in a subproblem for Plug-in Post and Weak Post adaptation is referred to as *postmatch driven synthesis* since the postconditions of the problem have been met. *Prematch driven synthesis* refers to synthesizing subproblems for adaptation using Plug-in Prematched components since the preconditions have been met.

5.1.1 Postmatch Driven Synthesis

A Weak Post or Plug-in Post matched component is abstractly represented by $\mathrm{Component}_B$ in the sequential architecture in Fig. 6. A subproblem specification must be synthesized in order to find a component, i.e., $\mathrm{Component}_A$, to adapt $\mathrm{Component}_B$. $\mathrm{Component}_A$ must change the environment to allow $\mathrm{Component}_B$ to execute and satisfy the behavior of the problem for all legal inputs. The subproblem synthesis is specified in Definition 2.

Definition 2. *Given a problem $P(D, R, I, O)$ and a Weak Post/Plug-in Post matched component $B(D_B, R_B, I_B, O_B)$, the synthesized subproblem for the missing functionality in the sequential architecture is:*

- *Domain:* D.
- *Range:* $D_B \cup \{r \in R | \neg\exists x \in R_B | \rho(x) \rightarrow r\}$.
- *Preconditions:* $\forall d : D | I(d)$.

```
// Sequential architecture solution to problem P1
problem P1() :: null is
  export all;
begin
  use negate;
  use sub;
  facet P1(a :: input real; b :: input real;
           c :: output real) ::
    state_based_semantics is
    export all;
    x__0 :: M__Type(sub.s);
  begin
    arch0: negate(b, x__0);
    arch1: sub(a, x__0, c);

    pre:   true;
    post:  c' = (a + b);
  end facet P1;
end package P1;
```

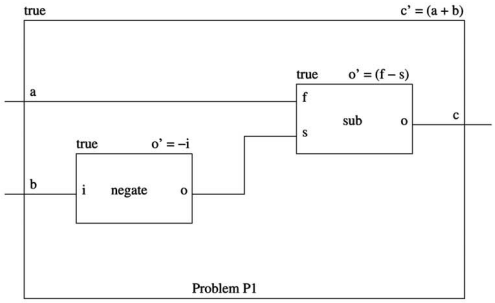


Fig. 10. Solution to problem P1 using the sequential architecture.

$$\dfrac{I_B \wedge O_B \Rightarrow O_P \qquad \dfrac{\dfrac{\dfrac{\dfrac{behConstraint1}{I_P \Rightarrow I_A}\ (4) \qquad \dfrac{\dfrac{behConstraint2}{I_P \wedge O_A \Rightarrow I_B}\ (6) \qquad \dfrac{behConstraint3}{I_P \wedge O_A \wedge O_B \Rightarrow O_P}\ (8)}{I_P \wedge O_A \Rightarrow \neg O_B \vee O_P}\ (7)}{I_P \wedge O_A \Rightarrow I_B \wedge (\neg O_B \vee O_P)}\ (5)}{(I_P \Rightarrow I_A) \wedge (I_P \wedge O_A \Rightarrow I_B \wedge (\neg O_B \vee O_P))}\ (3)}{(I_{synth} \Rightarrow I_A) \wedge (I_{synth} \wedge O_A \Rightarrow O_{synth})}\ (2)}{(I_B \wedge O_B \Rightarrow O_P) \wedge ((I_{synth} \Rightarrow I_A) \wedge (I_{synth} \wedge O_A \Rightarrow O_{synth}))}\ (1)$$

Fig. 11. Postmatch driven sequential synthesis inference tree.

- *Postconditions:* $\forall d : D, x : D_B, y : \{r \in R | \exists x \in R_B | \rho(x) \rightarrow r\}, r : R | I_B(x) \wedge (\neg O_B(x, y) \vee O(d, r))$.

In Fig. 11, the behavioral relationships specified in the sequential architecture theory is inferred from the match conditions. The inference starts from the Weak-Post (the Plug-in Post can also be inferred) match condition between the partially matched component and the problem, and the Satisfies match condition between the component for adaptation and the synthesized subproblem. In Step 1, the conjunction is split for simplification. The synthesized pre and postconditions are replaced with Definition 2 in Step 2, and the conjunction is split in Step 3. The first behavioral constraint of the sequential architecture follows immediately in Step 4. The conjunction in the consequent is split in Step 5, and the second behavioral constraint is satisfied in Step 6. Last, the negated term is moved to the antecedent in Step 7, which leads to the third behavioral constraint.

For illustration purposes, the simple math problem in Fig. 12 is queried using the small component library of simple math functions in Fig. 13. The library also shows the degree of match between a component and a problem as well as the port connections that were used to obtain the match condition. The library shows that the *pInc* component is a Weak Post match to problem *P2* using a bijective port connection. The synthesized subproblem using Definition 2 is specified in Fig. 14. The range of the subproblem consists of the input types to the component as well as any output types of the problem that were not instantiated during signature matching. Since all output ports of the problem were instantiated in this example, the range of the subproblem reduces to the input types of the component.

The subproblem in this example thus specifies:

$$\forall d : D, d_B : D_B, r : R | I(d) \Rightarrow I_B(x) \wedge (\neg O_B(x, r) \vee O(d, r))$$

```
// Simple math problem
package P2() :: null is
  export all;
begin
  facet P2(x :: input real;
           z :: output real) :: state_based_semantics is
    export all;
  begin
    pre:   true;
    post:  if (x < 0)
           then (z' = ((-1 * x) + 1))
           else (z' = (x + 1)) end if;
  end facet P2;
end package P2;
```

Fig. 12. Problem specification of a simple math problem.

or, more specifically:

$$x : real, a : real, z : real | (true \Rightarrow (a' >= 0)) \wedge (\neg(z = (a' + 1)) \vee (if(x < 0)\ then(z' = ((-1 * x) + 1))\ else(z' = (a + 1))\ endif)).$$

The variables `x` and `a` are the input and output of the synthesized subproblem, respectively, however the variable `z` is being referenced. This variable represents a quantified variable over the pre and postconditions. In order to avoid variable name conflicts, the variable names have been mapped to unique names, i.e., $P2.x : real \rightarrow P3.i\_\_0 : real$, $pInc.a : real \rightarrow P3.o\_\_0 : real$, and $P2.z : real \rightarrow P3.q\_\_0 : real$. The *absVal* component is the only component that completely satisfies the requirements of subproblem *P3*. Fig. 19 shows a solution to problem P2 using the sequential architecture.

5.1.2 Prematch Driven Synthesis

A component retrieved using the Plug-in Pre match condition can be abstractly represented as Component_A in the sequential architecture in Fig. 6. A subproblem specification has to be synthesized in order to find components, i.e., Component_B, to adapt Component_A. Component_B must change the results of Component_A to satisfy the behavior of the problem in all cases. The subproblem synthesis is defined in Definition 3. The behavior relationships specified in the sequential architecture theory are correctly inferred from the match conditions in Fig. 15.

Definition 3. *Given a problem $P(D, R, I, O)$ and a Plug-in Pre matched component $A(D_A, R_A, I_A, O_A)$, the synthesized subproblem for the missing functionality in the sequential architecture is:*

- *Domain:* $R_A \cup \{d \in D | \neg\exists x \in D_A | \rho(x) \rightarrow d\}$.
- *Range:* R.
- *Preconditions:* $\forall d : \{d \in D | \exists x \in D_A | \rho(x) \rightarrow d\}, x : R_A | I(d) \wedge O_A(d, x)$.
- *Postconditions:* $\forall d : D, r : R | O(d, r)$

Consider starting with the absVal component to solve problem P2. The synthesized subproblem using constraints from this component and problem P2 using the Definition 3 is specified in Fig. 16. The pInc component is retrieved to generate the same adaptation architecture solution in Fig. 19.

5.2 Alternative Adaptation

In the alternative architecture theory, Component_A can perform the same functionality as the problem, but only on a subset of the legal inputs. The problem would be solved if another component, namely, Component_B, performs the

```
// Positive constrained increment
package pInc() :: null is
  export all;
begin
  facet pInc(a :: input real;
             b :: output real) :: state_based_semantics is
    export all;
  begin
    pre:  a >= 0;
    post: b' = (a + 1);
  end facet pInc;
end package pInc;
```

(Bijective) Match on P1: (nil)
(1−1) Match on P1: (nil)
(Bijective) Match on P2: (Weak Post [(a −> x)(b −> z)])
(Bijective) Match on P3: (nil)
(Bijective) Match on P4: (Satisfies [(a −> i__0)(b −> o__0)])
(Bijective) Match on P5: (Weak Post [(a −> i__0)(b −> o__0)])
(Bijective) Match on P6: (nil)
(Bijective) Match on P7: (nil)

```
// Greater than component
package gt() :: null is
  export all;
begin
  facet gt(m :: input real; n :: input real;
           o :: output boolean) :: state_based_semantics is
    export all;
  begin
    pre:  true;
    post: o' = (m > n);
  end facet gt;
end package gt;
```

(Bijective) Match on P1: (nil)
(1−1) Match on P1: (nil)
(Bijective) Match on P2: (nil)
(Bijective) Match on P3: (nil)
(Bijective) Match on P4: (nil)
(Bijective) Match on P5: (nil)
(Bijective) Match on P6: (nil)
(Bijective) Match on P7: (nil)

```
// Real number subtraction component
package sub() :: null is
  export all;
begin
  facet sub(f :: input real;
            s :: input real;
            o :: output real) :: state_based_semantics is
    export all;
  begin
    pre:  true;
    post: o' = (f − s);
  end facet sub;
end package sub;
```

(Bijective) Match on P1: (Plug−in Pre [(f−>a)(s−>b)(o−>c)])
(1−1) Match on P1: (Plug−in Pre [(f−>a)(s−>b)(o−>c)])
(Bijective) Match on P2: (nil)
(Bijective) Match on P3: (nil)
(Bijective) Match on P4: (nil)
(Bijective) Match on P5: (nil)
(Bijective) Match on P6: (nil)
(Bijective) Match on P7: (Satisfies [(f−>i__1)(s−>i__0)(o−>o__0)])

```
// Great than equal to component
package geq() :: null is
  export all;
begin
  facet geq(m :: input real; n :: input real;
            o :: output real) :: state_based_semantics is
    export all;
  begin
    pre:   true;
    post: o' = (m >= n);
  end facet geq;
end package geq;
```

(Bijective) Match on P1: (nil)
(1−1) Match on P1: (nil)
(Bijective) Match on P2: (nil)
(Bijective) Match on P3: (nil)
(Bijective) Match on P4: (nil)
(Bijective) Match on P5: (nil)
(Bijective) Match on P6: (nil)
(Bijective) Match on P7: (nil)

```
// Absolute value component
package absVal() :: null is
  export all;
begin
  facet absVal(i :: input real;
               o :: output real) :: state_based_semantics is
    export all;
  begin
    pre:  true;
    post: o' = abs(i);
  end facet absVal;
end package absVal;
```

(Bijective) Match on P1: (nil)
(1−1) Match on P1: (Plug−in Pre [(i −> b)(o −> c)])
(Bijective) Match on P2: (Plug−in Pre [(i −> x)(o −> z)])
(Bijective) Match on P3: (Satisfies [(i −> i__0)(o −> o__0)])
(Bijective) Match on P4: (nil)
(Bijective) Match on P5: (nil)
(Bijective) Match on P6: (nil)
(Bijective) Match on P7: (nil)

```
// Numerical negation component
package negate() :: null is
  export all;
begin
  facet negate(i :: input real;
               o :: output real) :: state_based_semantics is
    export all;
  begin
    pre:  true;
    post: o' = −i;
  end facet negate;
end package negate;
```

(Bijective) Match on P1: (nil)
(1−1) Match on P1: (Plug−in Pre [(i −> b)(o −> c)])
(Bijective) Match on P2: (Plug−in Pre [(i −> x)(o −> z)])
(Bijective) Match on P3: (Plug−in Pre [(i −> i__0)(o −> o__0)])
(Bijective) Match on P4: (nil)
(Bijective) Match on P5: (nil)
(Bijective) Match on P6: (nil)
(Bijective) Match on P7: (nil)

Fig. 13. Small library of math components.

same functionality but covers the rest of the legal inputs. Since the pInc component satisfies problem P2 when the input is positive, the alternative adaptation tactic can be applied to find a component (or another architecture) that satisfies the rest of the problem when the input is (at least) not positive. The synthesis is defined in Definition 4. Using

```
// Post-match driven synthesis to problem P2 using
// component pInc and the sequential architecture
package P3() :: null is
  export all;
begin
  facet P3(i__0 :: input real;
           o__0 :: output real) :: state_based_semantics is
    export all;
    q__0 :: real;
  begin
    pre:  true;
    post: (o__0' >= 0) and
        ((not(q__0 = (o__0' + 1))) or
         (if (i__0 < 0)
          then (q__0' = ((-1 * i__0) + 1))
          else (q__0' = (i__0 + 1)) end if));
  end facet P3;
end package P3;
```

Fig. 14. Postmatch driven synthesis to problem P2 using component pInc and the sequential architecture.

$$\dfrac{\dfrac{\dfrac{behConstraint2}{I_P \wedge O_A \Rightarrow I_B}\ (5) \quad \dfrac{behConstraint3}{I_P \wedge O_A \wedge O_B \Rightarrow O_P}\ (6)}{(I_P \wedge O_A \Rightarrow I_B) \wedge (I_P \wedge O_A \wedge O_B \Rightarrow O_P)}\ (4) \quad \dfrac{behConstraint1}{I_P \Rightarrow I_A}\ (2)}{(I_P \Rightarrow I_A) \wedge ((I_{synth} \Rightarrow I_B) \wedge (I_{synth} \wedge O_B \Rightarrow O_{synth}))}\ (3)\ (1)$$

Fig. 15. Prematch driven sequential synthesis inference tree.

```
// Pre-match driven synthesis to problem P2 using
// component absVal and the sequential architecture
package P4() :: null is
  export all;
begin
  facet P4(i__0 :: input real;
           o__0 :: output real) :: state_based_semantics is
    export all;
    q__0 :: real;
  begin
    pre:  true and (i__0 = abs(q__0));
    post: if (q__0 < 0)
          then (o__0' = ((-1 * q__0) + 1))
          else (o__0' = (i + q__0)) end if;
  end facet P4;
end package P4;
```

Fig. 16. Prematch driven synthesis to problem P2 using component absVal and the sequential architecture.

this definition, the synthesized subproblem specifying the missing functionality to solve problem P2 using the constraints from pInc is shown in Fig. 17.

Definition 4. *Given a problem $P(D, R, I, O)$ and a Weak Post/Plug-in Post matched component $A(D_A, R_A, I_A, O_A)$, the synthesized subproblem for the missing functionality in the alternative architecture is:*

- *Domain: D.*
- *Range: R.*
- *Preconditions: $\forall d : D | I(d) \wedge \neg I_A(d)$.*
- *Postconditions: $\forall d : D, r : R | O(d, r)$.*

The behavioral constraints are inferred from the match conditions in Fig. 18. In Step 1, the synthesized pre and postconditions are replaced with Definition 4. Given that $I_P \wedge \neg I_A \Rightarrow I_B$, the equation is rewritten in Step 2. Step 3 involves the joining of the antecedents of sequents with O_P in the consequence. By splitting the conjunction in Step 4, the behavioral constraints become apparent.

The solution to the subproblem in Fig. 17 requires another adaptation architecture. It is clear to see that a

```
// Synthesis to problem P2 and component pInc using the
// alternative architecture
package P5() :: null is
  export all;
begin
  facet P5(i__0 :: input real; o__0 :: output real)
    :: state_based_semantics is
    export all;
  begin
    pre:  true and not(i__0 >= 0);
    post: if (i__0 < 0)
          then (o__0' = ((-1 * i__0) + 1))
          else (o__0' = (i__0 + 1)) end if;
  end facet P5;
end package P5;
```

Fig. 17. Synthesis to problem P2 and component pInc using the alternative architecture.

$$\dfrac{\dfrac{\dfrac{\dfrac{behConstraint2}{(I_A \wedge O_A) \vee (I_B \wedge O_B) \Rightarrow O_P}\ (5) \quad \dfrac{behConstraint1}{I_P \Rightarrow I_B \vee I_A}\ (7)}{I_P \wedge \neg I_A \Rightarrow I_B}\ (6)}{((I_A \wedge O_A) \vee (I_B \wedge O_B) \Rightarrow O_P) \wedge (I_P \wedge \neg I_A \Rightarrow I_B)}\ (4)}{(I_A \wedge O_A \Rightarrow O_P) \wedge (I_P \wedge \neg I_A \Rightarrow I_B) \wedge (I_B \wedge O_B \Rightarrow O_P)}\ (3)$$
$$\dfrac{(I_A \wedge O_A \Rightarrow O_P) \wedge ((I_P \wedge \neg I_A \Rightarrow I_B) \wedge (I_P \wedge \neg I_A \wedge O_B \Rightarrow O_P))}{(I_A \wedge O_A \Rightarrow O_P) \wedge ((I_{synth} \Rightarrow I_B) \wedge (I_{synth} \wedge O_B \Rightarrow O_{synth}))}\ (2)\ (1)$$

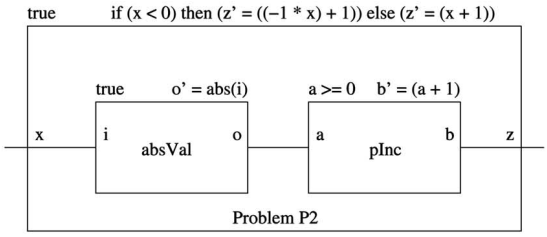
Fig. 18. Alternative synthesis inference tree.

sequential architecture using the negate and pInc components provides a solution. The limitation of the alternative architecture is that both components will drive the output of the solution, therefore a control mechanism needs to control which component will drive the output in the appropriate situation. For instance, pInc will generate a nonsensical output when the input is not positive. Expecting control structures to exist in the library is a significant drawback to the alternative architecture. Frequently it is sufficient to synthesize the language's control structures since the control structures are simple. The solution to problem P2 using the alternative architecture is shown in Fig. 19.

5.3 Parallel Adaptation

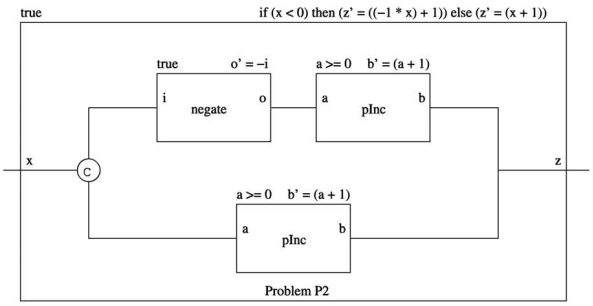
Parallel adaptation can be viewed from two different perspectives. In a bottom-up perspective, a component is first retrieved from the library and the architecture is evaluated based on the component's missing behavior. In a top-down perspective, the problem is first decomposed into independent subproblems. The components that match the subproblems are composed in a parallel architecture. Our framework uses the top-down approach in applying specification slicing for parallel adaptation [20].

Program slicing [30] is a decomposition process used to isolate a subset of program behavior. A program slice is a subprogram that contains only those statements and variables that affect or are affected by a slicing criterion. A slice criterion is a set of variables that are of interest at some point in the program. The goal is to use slicing to decompose a problem specification into independent slices, i.e., subproblems. The retrieval engine is then used to locate components that satisfy the slices. We have discussed specification slicing and presented an algorithm for parallel adaptation using slicing in other works [20].




```
// Sequential architecture solution
package P2() :: null is
  export all;
begin
  use absVal;
  use pInc;
  facet P2(x :: input real;
           z :: output real) ::
      state_based_semantics is
    export all;
    x__0 :: M__Type(pInc.a);
  begin
    arch0: absVal(x, x__0);
    arch1: pInc(x__0, z);
    pre:   true
    post:  if (x < 0)
           then (z' = ((-1 * x) + 1))
           else (z' = (x + 1)) end if;
  end facet P2;
end package P2;
```

(a)




```
// Alternative architecture solution
package P2() :: null is
  export all;
begin
  use negate;
  use pInc;
  facet P2(x :: input real;
           z :: output real) ::
      state_based_semantics is
    export all;
    x__0 :: M__Type(pInc.a);
    x__1 :: M__Type(pInc.a);
    x__2 :: M__Type(negate.i);
  begin
    arch0: if (x >= 0) then (x__1' = x)
           else (x__2' = x) end if;
    arch1: pInc(x__1, z);
    arch2: negate(x__2, x__0);
    arch3: pInc(x__0, z);
    pre:   true
    post:  if (x < 0) then (z' = ((-1 * x) + 1))
           else (z' = (x + 1)) end if;
  end facet P2;
end package P2;
```

(b)

Fig. 19. Solutions to problem P2 using the (a) sequential architecture and the (b) alternative architecture.

6 Evaluation

To evaluate our framework, we calculate four metrics. The metrics calculated are precision, recall, the average number of components per solution, and the average number of proof obligations (match condition proof) required to retrieve a set of results, on a collection of queries. *Precision* is defined as the ratio of correct solutions retrieved to the total number of results retrieved. High precision is the result of retrieving few irrelevant or invalid solutions. *Recall* is defined as the ratio of the number of correct solutions retrieved to the number of correct solutions that exist in the library. Ideally, recall should be high, meaning solutions should not be missed. Generally, the trade off between recall and precision is inversely proportional.

The equation for recall must be modified since a library can contain an infinite number of possible architectural solutions. For instance, a problem to increment an input value can be solved by an infinite number of solutions (constructed using N decrement components and N + 1 increment components, for all $N \geq 0$). Some of these solutions may contain nonsensical or redundant configurations of components, yet nevertheless solve the problem. We propose three different methods for calculating recall in Definitions 5, 6, and 7, each using different finite grouping relations.

Definition 5. *$Recall_1$ is defined as the ratio of the number of relevant component groups retrieved to the number of relevant component groups in the library. The grouping relation is defined as the containment of some combination (without replacement) of components such that a solution exists.*

Definition 6. *$Recall_2$ is defined as the ratio of the number of relevant component groups retrieved to the number of relevant component groups in the library. The grouping relation is defined as the containment of the smallest combination (without replacement) of components such that a solution exists.*

Definition 7. *$Recall_3$ is defined as the ratio of the number of relevant solutions retrieved to the number of relevant solutions in the library, where a solution contains a threshold of N components, where $N > 0$.*

6.1 Evaluation Library and Query Set

Evaluation is performed over a library containing 46 complex mathematical specifications, a library of 106 list manipulation specifications, a library of 30 record manipulation specifications, and a library consisting of 42 digital

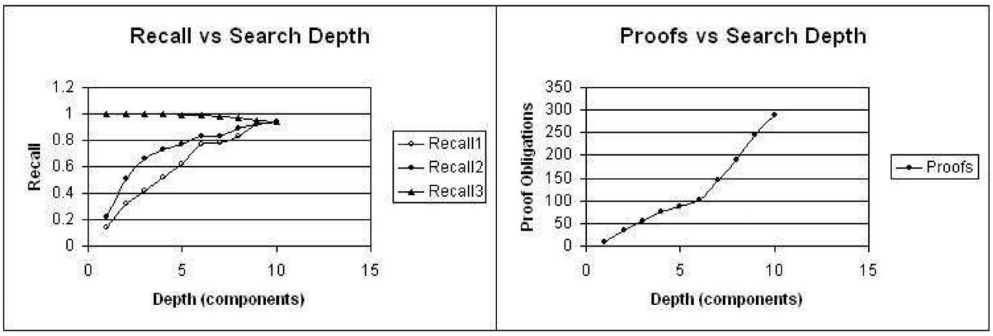


Fig. 20. Recall and proof obligations versus search depth.

signal processing specifications. These libraries have also been used in evaluating other works [23], [5], [21] and reflect the primary SPARTACAS application domains of embedded system and experimental data processing software. Although such systems are physically small, the costs associated with software development dominate production and maintenance costs making them ripe for software reuse. The methodology can be applied to larger libraries and other application domains, given that the adaptation architectures are applicable.

The test query set contains 103 queries, that can be classified into six types: queries for solutions that are solved

1. by one and only one component,
2. by a single component, but can also be solved by an architecture of components,
3. only by an architecture of components,
4. by an infinite number of architecture configurations,
5. by multiple subarchitectures, or
6. by components from multiple libraries.

The experiments are designed to test the automated adaptation of partial solutions using adaptation architectures; hence, the majority of queries are designed to be solved by an architecture. The experiments in other works using the same libraries focused on retrieval of single component solutions to queries; therefore, a direct comparison of results can not be performed.

6.2 Empirical Results

Our framework was implemented and evaluation results were obtained over the query set. The retrieval framework uses a depth-first architecture construction strategy. Since it is possible to get trapped in the construction of an architecture without an end, there must be a depth, or a limit on the number of components used in the construction of an architecture (solution size). The depth is equivalent to the number N in the definition of Recall_3. Fig. 20 shows the relationships between recall and the number of proof obligations versus the search depth. Recall_1 showed a fairly linear increase versus the search depth since there was a well distribution of solution sizes. Recall_2 initially had large gains, however, as the search depth was increased, there were diminishing returns. This result was expected since, based on the grouping definition of Recall_2, many solutions groups can be found with a small search depth. Recall_3 remained very high for small search depths, but slowly decreased as the search depth increased. This result is accredited to the theorem-prover's undecidability. As the depth is increased, the number of adaptations increase which commonly results in large and complex synthesized subproblems. The theorem-prover strategies fail to prove the resulting complex proof obligations in a reasonable amount of time.

The second graph shows an exponential increase in the average number of proof obligations per solution. As mentioned, as the depth is increased, complex subproblems are generated. Admittedly, it is increasingly rare to find a complete match to these subproblems, yet many partial matches are found, and therefore many more proof obligations are generated.

Fig. 21 shows the impact that the port connection methods have on recall (using Recall_2) and proof obligations. In terms of recall, the one-to-one and onto port connection methods had higher recall than the bijective port connection. The bijective port connection method requires matching components to have an equal number of input/output ports as the problem. The less restrictive port connection methods remove this restriction, thus finding more partial matches. The onto port connection method was comparable to one-to-one since both port connection methods were applicable for solutions to the query set. Obviously, the increase in recall comes at the expense of proof obligations when using a less restrictive port connection. This result, more so for the onto method, stems from the increase in signature combinations that must be tried than with the bijective method.

In all the experiments, SPARTACAS was able to maintain very high precision (almost always 100 percent), see Fig. 22. High precision was the result of using an automated theorem-prover to formally verify and maintain the constraints of the problem during adaptation and solution construction. Fig. 22 also depicts the relationship between search depth, port connection method, and the size of the solutions generated. In general, as the depth is increased, the average size of the solutions generated are larger. For reasons presented above, the relationship between size and port connection method is reflective of the solutions that can be discovered.

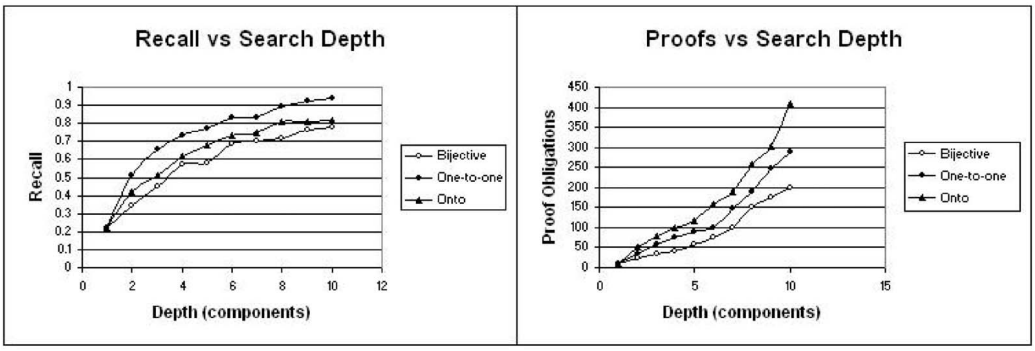


Fig. 21. Port connection effects on performance.

7 Related Work

ARBIE, developed by Chen and Cheng [3], provides a graphical environment to describe architectures. An architecture is composed of graphical elements (e.g., components and connectors), which contain a description of its properties. ABRIE uses a semiautomated capability to submit the architecture elements to a reuse engine. Existing components that match the properties of the elements can be reused and instantiated in the architecture. SPARTACAS automates the construction and instantiation of architectures to satisfy a design problem. The problem decomposition in parallel adaptation is similar to the approach in ABRIE.

Penix [23] presents a framework for automated component retrieval at the specification level. His work also suggested three adaptation architecture theories for component adaptation. In this work, we implemented an adaptation framework, using the architectures proposed by Penix, and experimental results were presented. To perform the adaptation, we precisely deduced and synthesized a subproblem to solve the adaptation requirements. Our work elsewhere [20] went into detail in specification slicing. In this work, we decompose a problem using slicing and address its application for performing parallel adaptation.

Purtilo and Atlee [26] have developed a system, called NIMBLE, that aides software designers by automating the adaptation of module interfaces. Adaptation of a module interface involves reordering, type coercion, and/or initializing or masking parameters. NIMBLE allows programmers to declare mappings and type adaptations of a program's interface, which get transformed into a module that encapsulates the desired adaptation. SPARTACAS uses a signature-based retrieval engine that automates the necessary reordering and type coercion in matching the interface of the problem to the interface of the component. The information is stored and used to properly instantiate the parameter configurations in the architecture specification.

Zhao [33] applies specification slicing for reuse-of-the-large using architecture description specifications. Using an architecture description language, a large specification describing a software system is sliced into a collection of smaller elements (components and connectors). Each element extracted can be reused in future designs. SPARTACAS uses specification slicing for reuse-of-the-small. Problems are decomposed into smaller subproblems such that matching components can be reused "as is" to solve the independent subproblems.

8 Future Work and Limitations

The architectures synthesized by SPARTACAS use shared-variable communication. Our work needs to address various communication protocols. Our framework can resolve this issue by: 1) including communication protocols as a search criteria when selecting components during retrieval, or 2) populating the component library with communication connector specifications [2] that can be retrieved and instantiated. The latter approach would

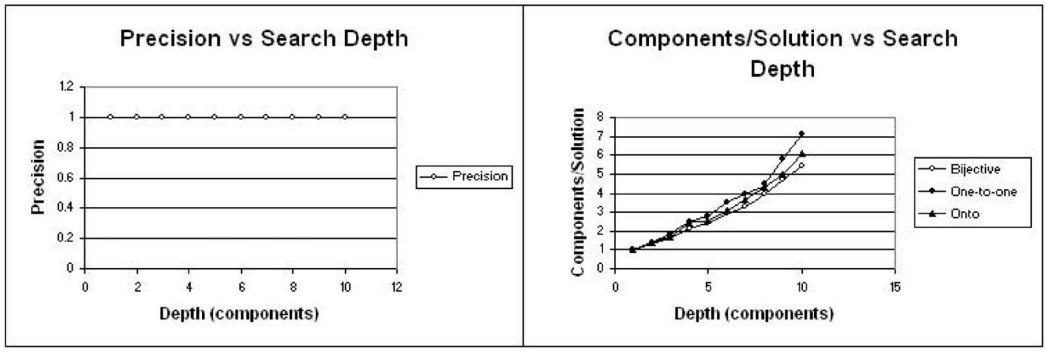


Fig. 22. Precision and solution size metrics.

maintain the generality of the component specifications and the SPARTACAS framework; however, it would increase the overhead in the retrieval engine due to the additional connector searches. Work also needs to be invested in applying reuse to different knowledge fields. Our reuse framework was evaluated on components in domains that used state-based semantics, where results are computed and placed on the output in the next state after the arrival of input.

SPARTACAS is limited in the type of adaptation architectures that it can synthesize. SPARTACAS is only capable of synthesizing three types of adaptation architectures (sequential, alternative, and parallel). Future work may include other architectures as tactics for adaptation. The current framework allows architectures, such as feedback architectures, to be stored in the library which can be retrieved and populated in a solution.

SPARTACAS is limited by the proving power of theorem-provers. There are proofs that are incapable of being solved, even with today's theorem-provers and processing power. The implementation is also limited by the depth that can be searched, SPARTACAS currently uses a depth-first search strategy in the architectural search tree. Other search strategies that estimate the cost to reach a solution could also be applied. SPARTACAS currently does not rank solutions since the ranking criteria is dependent on the features that the user wishes the solutions to possess. Allowing the user to specify such criteria (e.g., number of components used, propagation delay, extra technical information [9]) before retrieval can be added.

9 Conclusions

Reuse can potentially contribute many benefits to the software design cycle, but the costs associated with reuse must be reduced for reuse to become more common. A significant cost is the effort to search for and adapt components to satisfy a design problem. Most works have achieved efficient and effective component retrieval, but few works have concentrated on adapting partial matches. For adaptation to be feasible, the process needs to be reliable, error-free, scalable, and automated.

In this paper, we described a framework for automating component retrieval and adaptation for software reuse. We used a layered architecture using feature-based, signature-based, and specification-based retrieval engines to retrieve components that completely or partially match a problem. Three adaptation architecture theories for adapting the behavior of partially matching components were specified. Based on the relationship between the functionality of a partial match and the functionality of the problem to be solved, the missing functionality required to solve the rest of the problem is synthesized into a subproblem. The partially matched component is instantiated in the adaptation architecture with a component or architecture that satisfies the subproblem, thereby properly adapting the partial match. The framework was implemented and evaluated on examples from the embedded software and data processing domains. The results show SPARTACAS was able to recall approximately 94 percent of possible solutions while maintaining nearly 100 percent precision with adaptation.

References

[1] P. Alexander, D. Barton, and C. Kong, *Rosetta Usage Guide.* The Univ. of Kansas, ITTC, 2000.

[2] R. Allen and D. Garlan, "Formalizing Architectural Connection," *Proc. 16th Int'l Conf. Software Eng.,* pp. 71-80, May 1994.

[3] Y. Chen and B.H.C. Cheng, "Facilitating an Automated Approach to Architecture-Based Software Reuse," *Proc. 12th IEEE Int'l Conf. Automated Software Eng.,* pp. 238-245, Nov. 1997.

[4] D. Eichmann and K. Srinivas, "Neural Network-Based Retrieval From Software Reuse Repositories," *Neural Networks and Pattern Recognition in Human Computer Interaction,* R. Beale and J. Findlay, eds., pp. 215-228, Mar. 1992.

[5] B. Fischer and J. Schumann, "NORA/HAMMR: Making Deduction-Based Software Component Retrieval Practical," *Proc. CADE-14 Workshop Automated Theorem Proving in Software Eng.,* July 1997.

[6] B. Fischer and J. Schumann, "SETHEO Goes Software Engineering: Application of ATP to Software Reuse," *Proc. la Conferencia Anual de Ejecutivos (CADE-14),* July 1997.

[7] W. Frakes and C. Terry, "Software Reuse: Metrics and Models," *ACM Computing Surveys,* vol. 28, no. 2, pp. 415-435, June 1996.

[8] W.B. Frakes and C.J. Fox, "Sixteen Questions About Software Reuse," *Comm. ACM,* vol. 38, no. 6, pp. 75-87, June 1995.

[9] J.H. Gennari and M. Ackerman, "Extra-Technical Information for Method Libraries," *Proc. 12th Workshop Knowledge Acquisition, Modeling and Management (KAW'99),* Oct. 1999.

[10] M.R. Girardi and B. Ibrahim, "Using English to Retrieve Software," *The J. System and Software,* vol. 30, no. 3, pp. 249-270, Sept. 1995.

[11] A. Goldberg, "Reusing Software Developments," *Proc. Fourth ACM SIGSOFT Symp. Software Development Environments,* R.N. Taylor, ed., pp. 107-119, Dec. 1990.

[12] R.J. Hall, "Generalized Behavior-Based Retrieval," *Proc. 15th Int'l Conf. Software Eng.,* pp. 371-380, May 1993.

[13] C.A.R. Hoare, "An Axiomatic Basis for Computer Programming," *Comm. ACM,* vol. 12, pp. 576-580, 583, 1969.

[14] J.-J. Jeng and B.H.C. Cheng, "Specification Matching for Software Reuse: A Foundation," *Proc. ACM SIGSOFT Symp. Software Reuse,* pp. 97-105, Apr. 1995.

[15] C. Jones, "Economics of Software Reuse," *Computer,* vol. 27, pp. 106-107, July 1994.

[16] D.B.L.A. Kinley and D. Wilson, *Case-Based Reasoning: Experiences, Lessons, and Future Directions,* chapter "Learning to Improve Case Adaptation by Introspective Reasoning and CBR," AAAI Press/MIT Press, 1996.

[17] C.W. Krueger, "Software Reuse," *Computing Surveys,* vol. 24, pp. 131-183, June 1992.

[18] A. Mili, S. Yacoub, E. Addy, and H. Mili, "Toward an Engineering Discipline of Software Reuse," *IEEE Software,* vol. 16, no. 5, pp. 2-11, Sept./Oct. 1999.

[19] H. Mili, F. Mili, and A. Mili, "Reusing Software: Issues and Research Directions," *IEEE Trans. Software Eng.,* vol. 21, no. 6, pp. 528-562, June 1995.

[20] B. Morel and P. Alexander, "A Slicing Approach for Parallel Component Adaptation," *Proc. 10th IEEE Int'l Conf. and Workshop the Eng. of Computer-Based Systems,* pp. 108-114, Apr. 2003.

[21] M. Patil, "Soccer—A Specification Matching-Based Component Retrieval System," master's thesis, Univ. of Kansas, 2000.

[22] J. Penix and P. Alexander, "Toward Automated Component Adaptation," *Proc. Ninth Int'l Conf. Software Eng. and Knowledge Eng.,* pp. 535-542, June 1997.

[23] J. Penix and P. Alexander, "Efficient Specification-Based Component Retrieval," *Automated Software Eng.,* vol. 6, pp. 139-170, 1999.

[24] J. Penix, P. Baraona, and P. Alexander, "Classification and Retrieval of Reusable Components Using Semantic Features," *Proc. 10th Knowledge-Based Software Eng. Conf.,* pp. 131-138, Nov. 1995.

[25] R. Prieto-Díaz and P. Freeman, "Classifying Software for Reusability," *IEEE Software,* vol. 4, no. 1, pp. 6-16, Jan. 1987.

[26] J.M. Purtilo and J.M. Atlee, "Module Reuse by Interface Adaptation," *Software: Practice & Experience,* vol. 21, no. 6, pp. 539-556, June 1991.

[27] G. Sindre, R. Conradi, and E.-A. Karlsson, "The REBOOT Approach to Software Reuse," *The J. Systems and Software,* vol. 30, no. 3, pp. 201-212, Sept. 1995.
[28] D.R. Smith, "KIDS: A Semiautomatic Program Development System," *IEEE Trans. Software Eng.,* vol. 16, no. 9, pp. 1024-1043, 1990.
[29] B. Smyth and M.T. Keane, "Experiments on Adaptation-Guided Retrieval in Case-Based Design," Technical Report TCD-CS-94-17, Trinity College, Dublin, Dec. 1994.
[30] M. Weiser, "Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method," PhD thesis, Univ. of Michigan, Ann Arbor, 1979.
[31] A. Zaremski and J. Wing, "Signature Matching: A Key to Reuse," *ACM SIGSOFT Symp. the Foundations of Software Eng.,* Dec. 1993.
[32] A.M. Zaremski and J.M. Wing, "Specification Matching of Software Components," *Proc. Third ACM SIGSOFT Symp. the Foundations of Software Eng.,* Oct. 1995.
[33] J. Zhao, "A Slicing-Based Approach to Extracting Reusable Software Architectures," *Proc. Fourth European Conf. Software Maintenance and Reeng.,* pp. 215-223, 2000.

Brandon Morel received the BSCoE (honors) and MSCoE (honors) degrees from the University of Kansas, Lawrence, in 2000 and 2002, respectively. He is currently a PhD student at Washington University. His research interests include component retrieval, computer architecture, and formal methods.

Perry Alexander received the BSEE and BSCS degrees in 1986, the MSEE degree in 1988, and the PhD degree in 1992 from the University of Kansas, Lawrence. He was on the faculty at The University of Cincinnati for seven years where he established the Knowledge-Based Software Engineering Laboratory. He is currently an associate professor in the EECS Department and a principal investigator in the Information and Telecommunication Technology Center at The University of Kansas. His research interests include formal methods, specification languages, systems engineering, and language semantics. He is a senior member of the IEEE.